LLNL-TR-537251

LAWRENCE LIVERMORE NATIONAL LABORATORY

Intuitive visualizations through multi-domain projections for performance analysis at scale

A. Bhatele, P. T. Bremer, T. Gamblin, M. Schulz

March 12, 2012

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Intuitive visualizations through multi-domain projections for performance analysis at scale

Abhinav Bhatele, Peer-Timo Bremer, Todd Gamblin, Martin Schulz

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
E-mail: {bhatele, ptbremer, tgamblin, schulzm}@llnl.gov

Abstract

Performance analysis of parallel scientific codes is becoming increasingly difficult due to the rapidly growing complexity of applications and architectures. Existing tools fall short in providing intuitive views to reveal the root causes of performance problems. We have developed a new paradigm of projecting and visualizing performance data obtained from one domain onto other domains for faster, more intuitive analysis of applications. We gather performance data in three domains: hardware, application, and communication. For each domain, we define projections that allow the data to be viewed in the other domains. This framework is called the *HAC model*. The model not only allows us to directly compare the data across domains, but also allows us to use data visualization and analysis tools available in the other domains. Using the methodologies and visualization techniques mentioned above, we have demonstrated the careful unscrambling of otherwise tangled measurements caused by adaptive systems. By attributing performance measurements directly to their causes, we are able to visualize performance measurements in the domains most intuitive to the user, which are not necessarily those in which measurements are collected.

Keywords: data attribution, performance analysis, visualization, scalability, adaptivity, exascale

1 Introduction

Adaptivity is becoming ubiquitous across the high performance computing (HPC) software stack. At the application level, computational techniques such as adaptive mesh refinement (AMR) are used to reduce the cost of numerical solvers by increasing the solver resolution only for areas of the domain where higher resolution is needed [1, 2]. At a lower level, to ensure good performance, runtime libraries for adaptive codes dynamically redistribute the work generated by their refinement techniques among processors in a parallel system [3, 4]. Future exascale machines are expected to require adaptivity at even lower levels of the software stack, to distribute computational tasks among potentially heterogeneous compute resources, to balance power requirements and to adjust in case of hardware faults [15].

Adaptivity at any level presents a special challenge for users and developers of performance tools because it makes tying measurements to their root causes an even more difficult task. In a static domain decomposition, per-process performance measurements can be easily associated with specific application data. In adaptive codes such as AMR, a measurement on one process may potentially be associated with many pieces of the application data. Further, adaptive applications do not strictly follow the bulk-synchronous computational model traditionally used in HPC.

Interspersed with the computation and ghost exchange phases of traditional HPC applications, an adaptive code may reorganize work using sophisticated overlay networks and communication patterns. Performance measurements must therefore be carefully attributed to particular phases of computation and specific application data, in order to clearly identify the causes of observed performance problems.

Correctly attributing performance measurements to their causes requires understanding the complex relationships between different performance domains. We have developed a taxonomy of performance data that divides measurements into three key domains [17]. These are the hardware domain, consisting of processors embedded in a network with some topology; the application domain, comprised of information from the application's simulated physical domain; and the communication domain, comprised of abstract graphs with processes as nodes and communication between them represented as edges. We gather data in each of the three domains and we define projections of this data to the other two. This framework is called the *HAC model*. The model not only allows us to directly compare the data across domains, but also opens the door to using data visualization and analysis tools available in the other domains.

Using the methodologies and visualization techniques mentioned above, we have demonstrated the careful unscrambling of otherwise tangled measurements caused by adaptive systems. By attributing performance measurements directly to their causes, we are able to visualize performance measurements in the domains most intuitive to the user, which are not necessarily those in which they are measured. We have applied these techniques to several application codes and libraries: Miranda [5], a higher order hydrodynamics code; QBall, an implementation of First-Principles Molecular Dynamics; *hypre* [7], an library for solving sparse linear systems; and SAMRAI [11], a highly scalable structured AMR library used extensively in several large-scale DOE applications. In the context of SAMRAI, we used insights gained from projecting data on the communication domain to perform targeted optimizations and to improve its performance. Results for a 65,536-core run on a Blue Gene/P system show performance improvements of nearly 16%.

2 The HAC model

Most traditional scientific data analysis techniques are concerned with many quantities of interest (temperature, pressure, etc.) defined on a single common domain such as physical space. Performance data has many "natural" domains describing different aspects of application or system behavior. For example, hardware counter metrics are measured on a per-core or per-socket basis, and communication patterns are tracked within the MPI rank space abstraction. Understanding the meaning of these metrics typically requires relating them to other domains. For example, the performance of a particular core provides little information without a description of the part of the application domain to which it is assigned, or information about the portion of global communication it performs.

The *HAC* model allows tools to account for the multi-domain nature of this problem by identifying three key domains for performance measurements and by defining projections between them. Figure 1 illustrates this approach: we gather data in each of the three domains in the form of application data, performance metrics and communication profiles, and we define projections of this data to the other two. This not only allows us to directly compare the data across domains, but also opens the door to using data visualization and analysis tools available in the other domains. We can now apply scientific visualization tools to performance data projected into the application domain, or we can apply graph-based techniques to the same data projected into the communication domain. The *HAC* model provides a structured characterization of common performance analysis and optimization challenges in terms of relevant source data/domains, the necessary mappings, and the subsequent analysis requirements. This structure will lead to new insights, more focused research directions, and a common framework to describe, compare, and combine different approaches. In the following, we briefly describe the three data domains and a set of mappings among them using, as a reference, the combustion code S3D [10] (see Figure 2).

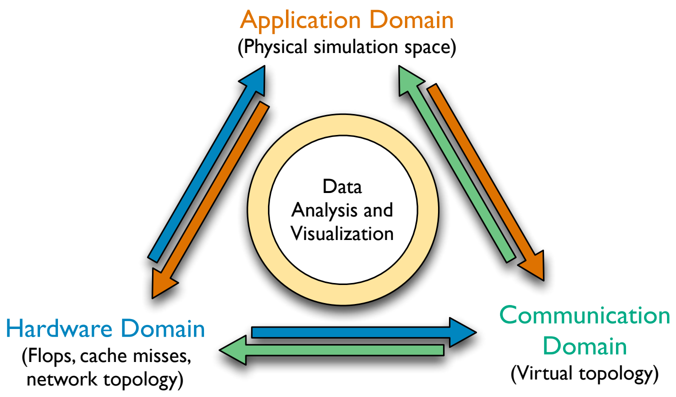

Application Domain
(Physical simulation space)
Data Analysis and Visualization
Hardware Domain
(Flops, cache misses, network topology)
Communication Domain
(Virtual topology)

Figure 1: The HAC model

Application Domain: The application domain represents the physical or other simulated system modeled by an application code. Often it is a piece of physical space with materials represented by some grid or mesh, but more abstract domains such as sparse matrices or large, abstract graphs are also possible. This is the domain most familiar to scientists, and it generally provides the mental reference frame during the code design stage. The application is ideal for exploring performance problems related to specific physical phenomena of the simulation. As an example, consider S3D, which describes a flame with some features of interest, as shown at top in Figure 2. Correlating features of the flame with performance data, such as unusually high cache miss counts, may highlight problems with a specific chemistry kernel or with some data structure in the application. Once we map the data into this space, we can exploit the large number of scientific visualization and analysis tools that already exist for application data.

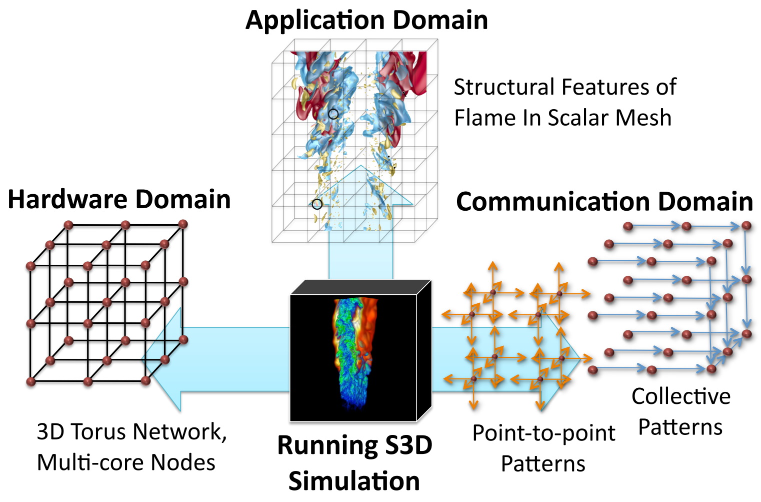


Figure 2: Domain decomposition of S3D

Hardware Domain: The hardware domain describes the physical hardware of a computer system in terms of computing cores and network links. A common configuration in today's integrated HPC systems is a mesh or torus network of multi-core compute nodes as shown on the left in Figure 2. Commodity clusters often rely on Infiniband (IB) networks with fat tree topologies. Each of these networks has its own graph of hardware links connecting nodes.

Data analysis in the hardware domain poses several challenges. Clearly, large scale graph algorithms such as clustering can be helpful. However, many of the largest supercomputers use regular grids connected into three-dimensional tori. Future machines will use higher-dimensional mesh/torus topologies. Given the large number of nodes and the Cartesian scaling properties, it is reasonable at scale to treat such a regular grid as a discrete representation of a continuous space. This allows the use of new classes of algorithms such as local correlation analysis [12, 16] or topological techniques [13, 9] to detect and extract features from high dimensional manifolds.

Communication Domain: The communication domain consists of a general graph. For MPI programs, such as the ones studied here, nodes correspond to MPI processes and edges correspond to message exchanges between them. During execution, most MPI applications will exhibit multiple distinct communication patterns. For example, as shown on the right of Figure 2, the S3D code uses primarily a stencil-based neighbor exchange pattern resulting in a grid that is almost identical to the hardware domain. This is interleaved with global collective patterns, such all-to-alls and the reduction shown in the figure. The projections of the communication patterns on to the hardware domain can provide useful insights into the efficiency of the communication phases in an application.

Root causes of performance problems lie in one of these domains described above, but their symptoms may lie in another. We propose new techniques for visualizing and analyzing performance data that correlate symptoms to causes by *projecting* performance data from one domain to another in order to make correlations and root causes more clear. Further, by projecting data from one domain to another, we can then use the set of data analysis techniques available in the destination domain to extract and compare features. For example, we might project floating point operation counts from the hardware domain into the application domain and then analyze them by applying feature-finding algorithms there. The correlation of hardware, communication, and application data allows application developers to understand how structures in their simulated domains correspond to real performance problems, and it also gives systems engineers a new, more intuitive perspective on how simulations map to physical hardware. This correlation is not possible with existing approaches.

The difficulty of projecting data across domains depends on how much and how frequently the relationships between domains change. For example, in a statically decomposed, structured grid application, the domain decomposition is fixed, and we can assume that per-process measurements are associated with a particular chunk of the decomposed application domain. In an AMR code, however, the physical domain is decomposed into variably sized units, which may move from process to process at runtime. We must therefore take special care to track the units as they move within the system in order to detect performance problems and attribute them to particular neighborhoods of the application domain.

Similarly, for structured grid codes, most communication is regular. For example, many such codes use a simple stencil-patterned ghost exchange among neighboring processes. We can easily make assumptions about which processes communicate and how much data will pass over each communication link. However, if there are many phases with substantially different communication patterns, we cannot attribute all bandwidth to the same algorithm. Instead, we must provide a more fine-grained analysis that can distinguish phases and track many independent communication patterns. In adaptive applications like AMR, this kind of dynamic behavior is driven by the application, its domain decomposition, and its phase structure. The behavior also changes depending on the particular problem being simulated. Performance measurement tools must therefore be able to map performance measurements pertaining to communication and computation back to entities in the application domain in order to find the root causes of performance problems.

3 Projecting data in the application domain

Performance counters and other measurement devices present on most modern microprocessors can provide data in the hardware domain. For parallel applications, the data is typically associated with the rank of the MPI process taking these measurements, as this is a readily available process identifier and MPI ranks roughly correspond to measurements for single cores. The MPI rank space is unintuitive, since it is a flat namespace with no inherent locality semantics. Figure 3 shows a simple plot of FP counts per MPI process for Miranda [5], a higher order hydrodynamics code. The plot shows that there is variable load, but otherwise provides no insight into application behavior.

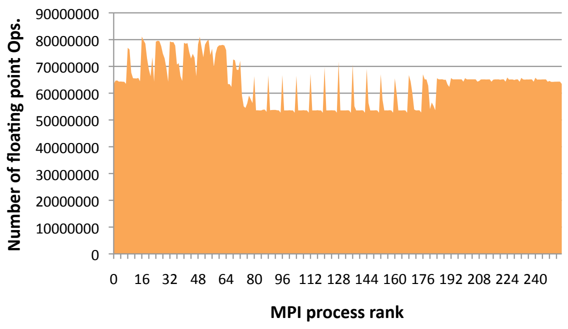
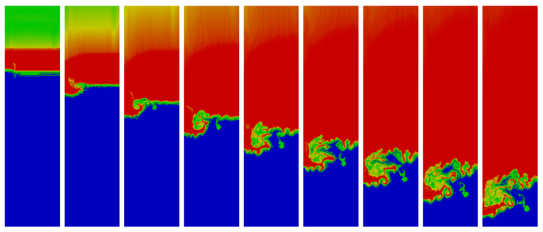


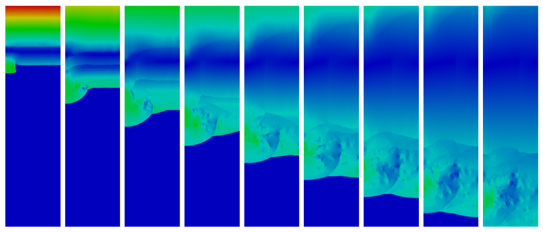
Figure 3: Number of floating operations per process for one time step of Miranda

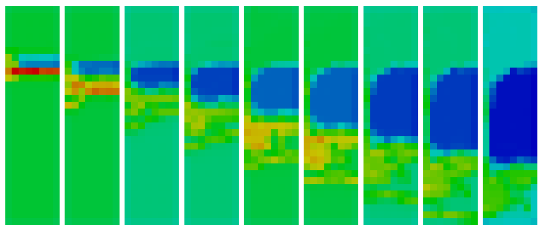
In order to provide better attribution of hardware measurements to application-domain structures, this first case study maps performance data from the hardware domain to the application domain. In our experiments, we simulate an ablator driving a shock into an aluminum section containing a void and producing a jet of aluminum. The simulation results of a 12 hour run using a 2D grid running on 256 cores of Hera (an Opteron-Infiniband cluster) are depicted in Figures 4(a) and 4(b) using nine selected time steps in regular intervals sorted from left to right. The figures clearly show the aluminum jet on the left side as well as the created shock wave traveling from top to bottom.

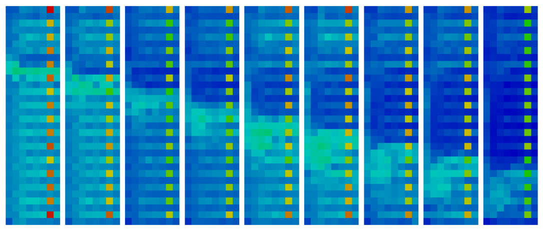
For this experiment, we concentrate on per-core performance information. We measure the total compute time spent per time step as well as three marquee performance counters: L1 miss rate, number of floating point operations, and number of branch miss-predictions. We measure this data using PAPI [14] from within a P^{N}MPI module loaded transparently at application start. We take advantage of the application's built-in visualization capability, and we write out our performance data as additional fields within periodic visualization dumps. The simulation data is organized in a 2D regular, dense grid, which is split into equal chunks in all dimensions and distributed among the processors in row major order. In our case study, we use 256 cores split into an 8 $\times$ 32 grid. We extract the mapping of grid cells to processor cores from the application, and we store the matching processor grid coordinates in the output files. Then, we use this data to determine which parts of the simulation grid are computed by which cores, and this provides us with the necessary projection function between the hardware and application domains.

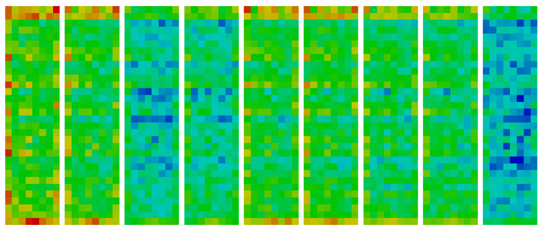
Plots of the four performance metrics are shown Figure 4(c)-(f). Despite the simplicity of the experiment, the use of the *HAC* model provides valuable insight not possible using MPI rank space alone. The projections of both FP operation counts and L1 cache misses clearly show features that mirror the movement of the physical shock wave through 2D space. By simple visual inspection we see that the shock wave itself requires a higher number of FP operations to compute, but in the wake of the wave we see significantly fewer operations per iteration. Furthermore, while the areas not affected by the shock wave show a steady number of operations, the computation of the wave clearly shows more variation. Similar observations, although with smaller differences, can be made for the L1 cache miss counts. Combined, these observations allow predictions of load imbalance.


(a) Aluminum distribution.


(b) Velocity distribution.


(c) Floating point operations.


(d) L1 cache misses.


(e) Branch miss-predictions.

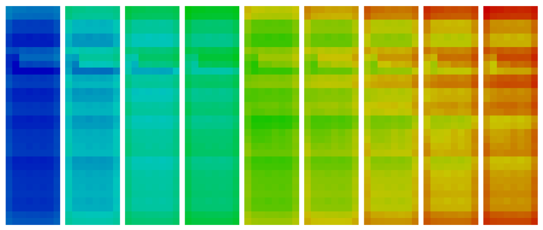

(f) Compute time per time step.

Figure 4: Simulation results (top row) and performance data mapped to the application domain (middle and bottom row) — left to right: simulation progress, blue shows low and red high values.

The number of branch mispredictions shows a different picture. We see higher numbers on the boundary of the domain caused by specialized code to handle boundary conditions. Over time, this effect is reduced, showing that the branch predictor was able to learn the application's behavior. The actual compute time per iteration (which includes time spent blocked within an iteration and is not a good measure for load balance) also shows increased computation time at the top and bottom boundary conditions. More importantly, we clearly see a gradual increase in runtime per iteration caused by the increasing complexity of the simulated problem.

4 Projecting data in the communication domain

In this section, we demonstrate the usefulness of projections in pinpointing a previously elusive scalability problem within the load balancer of SAMRAI [11, 8], a highly scalable structured AMR library used extensively in several large-scale DOE applications. Figure 5 shows the times spent by individual MPI processes in each sub-phase of load balancing on 256 cores of Intrepid (a Blue Gene/P system). The three sub-phases, load distribution, mapping generation, and overlap update are referred to as phase 1, 2 and 3 respectively in the rest of the paper. We can see that several processes are actually spending a significant amount of the total load balancing time in phase 1 (tall red bars). This does *not* show up in a traditional profiler because the results are presented in aggregate, and we lose information about behavior of the smaller number of slow processes in phase 1. We hypothesize that processes that have a larger fraction of the time spent in phase 2 or 3 might be waiting for processes stuck in phase 1, *i.e.*, load distribution.

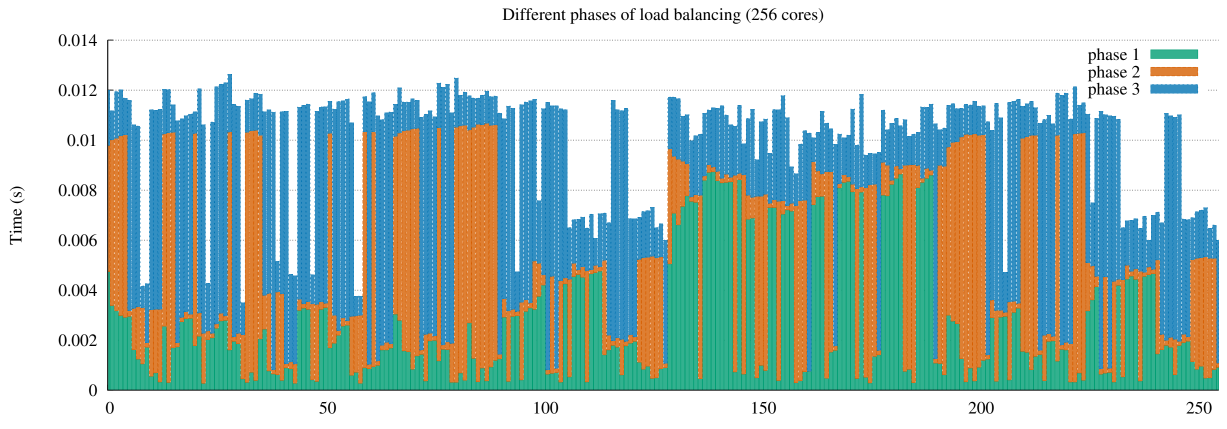


Figure 5: Time spent in different load balancing sub-phases for 256 MPI processes on Blue Gene/P.

Plotting timing information for different phases against a linear ordering of MPI processes by their ranks only gets us so far and is also unintuitive to the end user, since rank order and its mapping to compute resources is arbitrarily imposed by the MPI library and the scheduler. We must therefore map this information to another domain and visualize it there to truly understand it implications. SAMRAI communicates load variations and work loads (boxes) along a virtual tree network. To understand the communication behavior, we look at projections of phase timing data onto the communication domain, *i.e.*, the load balancing tree. We construct a pairwise communication graph among the MPI processes for the load balancing phase, which looks like a binary tree, and we color nodes by the time they spend in different sub-phases of the load balancer.

Figure 6 (left) colors the nodes in the tree network by the initial load on each process before load balancing starts. We see that loads of individual nodes are randomly distributed over the tree and do not appear to be correlated to the phase timings in the left diagram. However, the total loads for each of the four sub-trees give us some indication of what we'll find next. Three of the four sub-trees (in various shades of blue in the left diagram) have 2.83, 2.87 and 3.1% more load than the average whereas one (the sub-tree in red) has 9% less load than the average. This suggests that load has to flow from three overloaded sub-trees to the underloaded one to achieve load balance.

Figure 6 (right) shows the tree network used in the load balancing phase with each node colored by the time the corresponding MPI process spends in phase 1, *i.e.*, load distribution. Interestingly, in this view, we see that a particular sub-tree in the virtual topology or communication graph is

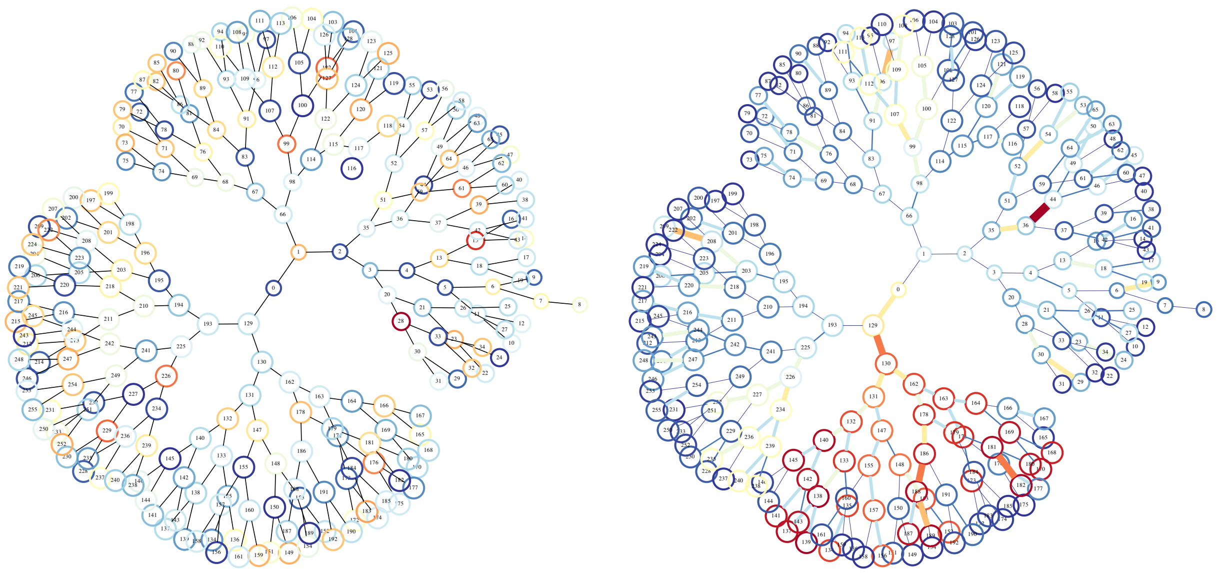


Figure 6: Phase timing data and load information visualized in the communication domain – the virtual tree network of the load balancing phase (256 processes on Blue Gene/P). On the left, the nodes are colored by the time spent in phase 1 *i.e.*, load distribution. On the right, the nodes are colored by the load in terms of the number of cells.

colored in orange/red, highlighting the processes that spend the most time in phase 1. Further, from mpiP output, we were able to ascertain that nearly 85% of this time is spent in an `MPI_Waitall` where a child is waiting to receive boxes from its parent. The problem escalates as we go further down this particular sub-tree, which is reflected in the increasing color intensity, i.e., processes farther away from the root spend longer time in this phase. Since we established that processes in one sub-tree are waiting for their parent to send them load, we color and weight each edge by the number of boxes that the edge's child node receives from the parent. We can see a flow bottleneck from node 0 to node 129 and from node 129 to node 130 near the top of the slowest sub-tree. This flow problem [6] is the cause of scaling issues in the load balancing phase of SAMRAI.

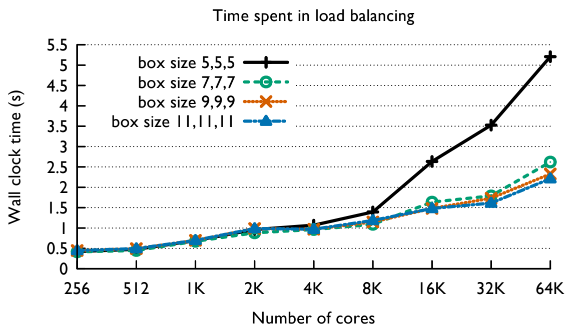


Figure 7: Improvement in load balancing time by making fewer boxes.

As initial steps and proof of concept, we reduce the amount of data sent around the tree by reducing the number of boxes being sent around by increasing the size of each box in terms of the number of cells it holds. The default value for the box size is (5, 5, 5) cells. We ran experiments with three larger box sixes and recorded the maximum number of boxes sent on any edge along with the timing information. Changing the box size leads to a reduction in the amount of traffic on the overlay network, which translates into a reduction of the time spent in load balancing (see Figure 7). Compared to the default box size, using (7, 7, 7) boxes, load balancing is completed in nearly half the time on 65,536 cores. This leads to an overall performance benefit of more than 16% by creating slightly larger boxes.

5 Conclusion

This paper presents the *HAC* model, which provides a structured approach to leverage the multi-domain nature of performance data. By collecting performance data in three key domains and describing projections of data between them, our framework provides new intuitive ways to analyze and visualize performance data. By extending and combining maps and analysis techniques in different domains, we showed that the *HAC* model enables users to differentiate between application-specific and system-specific performance problems.

We have described a structured framework to guide future research and highlight underdeveloped or missing components of a complete performance analysis toolkit. We are currently working to develop new types of scalable, automated analysis techniques using the foundational techniques presented here. We believe that the processes outlined in this paper and the visualization techniques presented are generally applicable and specially useful for adaptive scientific codes.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TR-537251).

References

[1] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82:64–84, May 1989.

[2] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[3] R. K. Brunner and L. V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[4] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–11. IEEE, 2007. Best Algorithms Paper Award.

[5] A. Cook, W. Cabot, M. Welcome, P. Williams, B. Miller, B. de Supinski, and R. Yates. Tera-scalable Algorithmms for Variable-Density Elliptic Hydrodynamics with Spectral Accuracy. In *Proceedings of IEEE/ACM Supercomputing '05*, November 2005.

[6] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *Information Theory, IRE Transactions on*, 2(4):117 –119, december 1956.

[7] R. Falgout, J. Jones, and U. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 267–294. Springer-Verlag, 2006.

[8] B. T. Gunney, A. M. Wissink, and D. A. Hysom. Parallel clustering algorithms for structured amr. *Journal of Parallel and Distributed Computing*, 66(11):1419 – 1430, 2006.

[9] A. Gyulassy, P.-T. Bremer, V. Pascucci, and B. Hamann. A practical approach to Morse-Smale complex computation: Scalability and generality. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1619–1626, 2008.

[10] E. R. Hawkes and J. H. Chen. Direct numerical simulation of hydrogen-enriched lean premixed methane–air flames. *Combustion and Flame*, 138:242–258, 2004.

[11] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.

[12] H. Jaenicke, A. Wiebel, G. Scheuermann, and W. Kollmann. Multifield visualization using local statistical complexity. *IEEE Transactions on Visualization and Computer Graphics*, 13:1384–1391, 2007.

[13] D. Laney, P.-T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Trans. Vis. and Comp. Graphics*, 12(5):1052–1060, 2006.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP User Group Conference*, June 1999.

[15] V. Sarkar. Exascale software study: Software challenges in extreme scale systems. Technical report, 2009.

[16] D. Schneider, A. Wiebel, H. Carr, M. Hlawitschka, and G. Scheuermann. Interactive comparison of scalar fields based on largest contours with applications to flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14:1475–1482, November 2008.

[17] M. Schulz, J. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci. Interpreting performance data across intuitive domains. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 206–215, September 2011.